Negative Learning Rates and P-Learning

Devon J. Merrill
Department of Computer Science and Engineering
UC San Diego
devon at ucsd dot edu

Abstract

We present a method of training a differentiable function approximator for a regression task using negative examples. We effect this training using negative learning rates. We also show how this method can be used to perform direct policy learning in a reinforcement learning setting.

1 Regression and Learning Rates

The goal of regression analyses is to find a regression function, a function that models the relationship between the independent variables (the inputs) and the dependent variables (the outputs). When a complex relationship between these variables, an exact regression function is not sought. Instead an approximate regression function is used to model the relationship.

Function approximators have been shown to be a sound method for finding approximate regression functions. Function approximates are generally trained using example input-output pairs from the function that is to be approximated. Given the input from the pair, the output of the function approximator is compared to the actual output from the example. The function approximator is then modified –possibly through gradient descent– so that its output matches the output in the example. This is one training step.

The function approximator is not usually modified so vigorously that the output matches the example output perfectly for each training. Extreme modification of the approximator tends to result in the loss or forgetting of the previous examples, that is, the previous training is overwritten by too strong of an update. Also, updates that are too vigorous tend to negatively affect the approximator's ability to generalize to unseen examples. Another issue occures when a function approximator is updated, the parameters of the model, it are changed. If these parameters are changed too quickly, they can overrun a computer's ability to represent these numbers. This is sometimes known as model explosion and it prevents the use of the function approximator.

To solve these and other problems, the updates to a function approximator are attenuated by a fractional amount, conceptualized as the learning rate. This allows the approximator to be pushed in the desired direction by a small, tunable amount. All approximators trained by gradient descent use a learning rate.

Numerous methods have been developed to automatically compute learning rates. Many use the second derivative of the parameter change (where the first derivative is the raw magnitude and direction of the update. This basic method is used from the familiar Newton's Method, to esoteric concave optimization methods.

Other methods use previous update amounts to tune the learning rate.

However, the most common method for setting a learning rate is to start with 0.1 and hope for the best.

2 Learning Rates as a Learning Channel

In regression, the learning rate does not contain information about what the approximator should output for any given input. This information is contained in the input-output example pairs. This is why the learning rate is always positive; a positive learning rate means that the approximator should match this example more closely.

Still, there are many situations where we do not have access to good input-output pairs. The pairs we have access to might not be from the actual function we are trying to model, but from any other function. The pairs we have might be totally random.

However, we might have access to a measure of how closely the output we have matches up with hypothetical output, given some input, from the function we are trying to approximate.

For example, take input-output pair $e = (x, z)$ where x is some input vector and z is an output vector. We would like our function approximator $nn(x)$ to give us the correct output from our target function $nn^*(x) = y$, but we do not have access to any example $nn^*(x) = y$ for any x. However, if we have a distance to desired output function function $dist^*(x, z) \rightarrow \mathbb{R}^1$ that gives a similarity measure between z and $nn^*(x) = y$, we can still train $nn(\cdot)$

We would like to use $dist^*(\cdot, \cdot)$ as a training signal to train n. Learning rate is useful as a channel for this distance to desired output function training signal.

To use the learning rate as a learning channel we can take each example in the training set and use it to assign a custom learning rate for each example. We should still have a global learning rate μ. Now, when we train on example i, the learning rate r_i for example e_i is $\mu \times dist^*(x_i, z_i)$.

Now we train a simple 2-layer artificial neural network to reproduce the sine function on the interval $[-5, 5]$ with 40 example points. The network uses $tanh$ activation function for the 128 hidden units. Each example is of the form $e_i = (x_i, z_i, r_i)$ where $x_i = -5 + i \times 0.25$ and z_i is drawn from a uniform distribution on the interval $[-1, 1]$. The value r_i is the learning rate factor for example i calculated by the distance to desired output function

$$dist^*(x_i) = r_i = |sin(x_i) - z_i|.$$

Let's see how this works in figure 1.

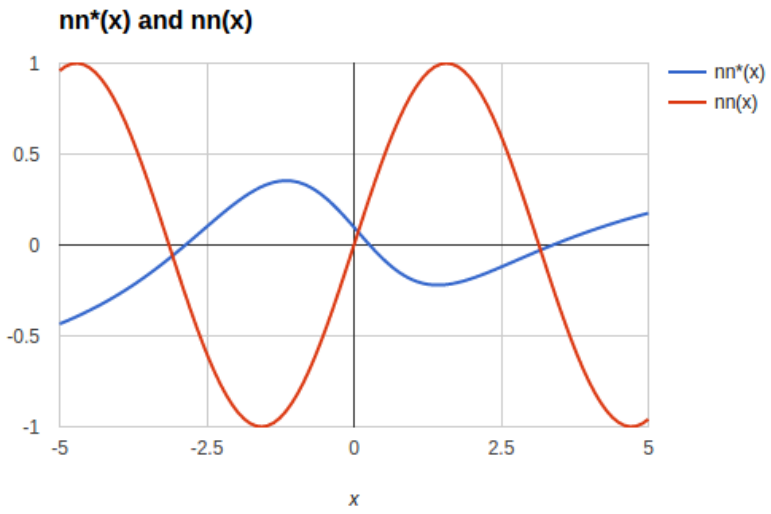

Figure 1: Output of network trained with per example learning rates $r_i = dist^*(x_i, z_i)$.

Well, obviously this is wrong, but it is doing something. What we actually want is

$$r_i = -\frac{dist^*(x_i, z_i) - min_i(dist^*(x_i, z_i))}{max_i(|dist^*(x_i, z_i)|)}.$$

This will give use a per example learning rate that decreases with distance to the target value and normalize to the range $[0, 1]$.

Let us see how that works with figure 2.

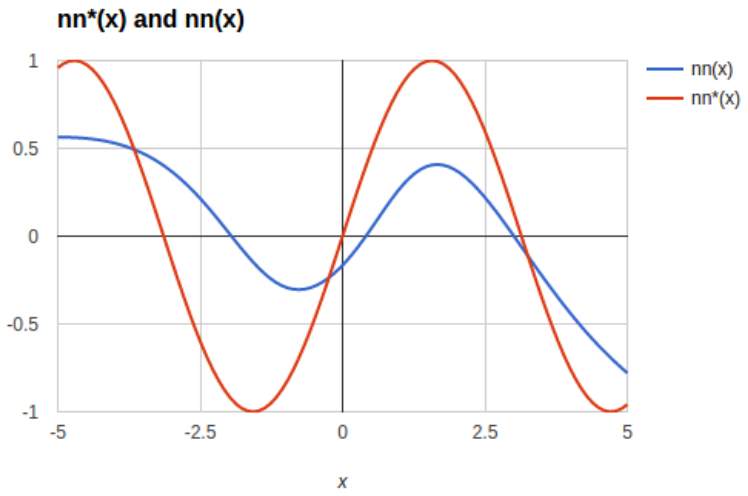

Figure 2: Output with per example learning rates $r_i = -\frac{dist^*(x_i,z_i)-min_i(dist^*(x_i,z_i))}{max_i(|dist^*(x_i,z_i)|)}$.

Now we seem to be getting somewhere. Let us now explore some intuition as to why this works.

2.1 Intuition

Normally when we are training a model we assume all the examples are drawn from the target distribution. That is why we can use a per example learning rate of $\mu \times 1$, we want the model to match these examples completely. If, instead, we know that the i training examples (x_i, z_i) are not from the target function, and instead are some distance to desired output $dist * (x_i, z_i)$ away, we do not want the model to match those examples exactly. We want it to match each example partially by some amount that is correlated to $dist^*(x_i, z_i)$.

If the distance to desired output $dist^*(x_i, z_i) = 0$ then we want it to match completely; the example is from the target distribution. If the distance to desired output $dist^*(x_i, z_i) > 0$ then the amount we want to match the example does down. This is where we get the $-dist^*(x_i, z_i)$ part of the formula.

Also, we assume that we have tuned the global learning rate μ to some amount that we do not want to go over. This is why we normalize the list of per example learning rate factors to give us

$$-\frac{dist^*(x_i, z_i) - min_i(dist^*(x_i, z_i))}{max_i(|dist^*(x_i, z_i)|)}.$$

With these transformed per example learning rate factors we match the closest examples completely while not matching the worst examples at all.

3 Negative Learning Rates

The network shown in figure 2 seems to be trying to match the target function, but it is still pretty bad. This is because we have set the learning rates of the worst examples to close to 0. Because of this we don't learn anything from the worst examples. To really match the target distribution we need to use all the examples given, not just the best ones. We can do this by using negative learning rates.

Negative learning rates are a little unusual, so we will try to give them a good treatment.

To get our negative learning rates we can constrain not to the range $[0, 1]$ but to the range $[-1, 1]$. The per example learning rates for the n examples become

$$r_i = -\frac{dist^*(x_i, z_i) - \frac{\sum_j z_j}{n}}{max_i(|dist^*(x_i, z_i) - \frac{\sum_j z_j}{n}|)}.$$

Now the worst examples have a learning rate factor of -1 and the best have a learning rate factor of 1. The examples of average distance to desired output function have a learning rate factor of 0, we do not learn anything from them, which seems to make sense intuitively.

With this change, let us see what happens in figure 3.

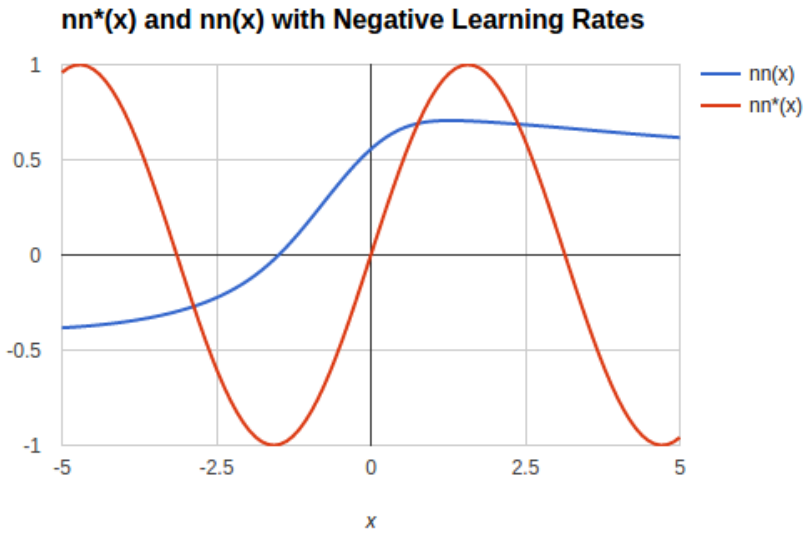

Figure 3: Output with per example learning rates $r_i = -\frac{dist^*(x_i,z_i)-\frac{\sum_j z_j}{n}}{max_i(|dist^*(x_i,z_i)-\frac{\sum_j z_j}{n}|)}$.

Unfortunately this does not work that well. We can understand why if we look more closely at the way training examples are presented to the network through the loss function.

We have been using the sum squared error (SSE) loss function to calculate the gradient being back-propagated through the network. This loss function has a minimum when the output of the network matches the example output, while it has a larger value the farther away the network's output is from the target output. When all the examples are good, this is the behavior that we want; training does not fix what is not broken, while training increases the further the output is from the desired output. However, when the examples can be negative, this behavior breaks the training.

For negative, "bad", examples, we want stronger training when the network output is close to the presented output. Then we can reduce training the farther away the network output is from the example output. This is the inverse of the normal behavior on positive examples.

To fix this problem we can replace the incoming gradient g to the network with $1/g$ when the learning rate is negative. This is equivalent to using a modified per example loss function which takes an additional parameter r_i:

$$L_i = \begin{cases} r_i \frac{(nn(x_i)-z_i)^2}{2} & r_i > 0 \\ r_i \, log(|nn(x_i) - z_i|) & r_i < 0 \end{cases}$$

The output of our network with this training scheme is shown in figure 4.

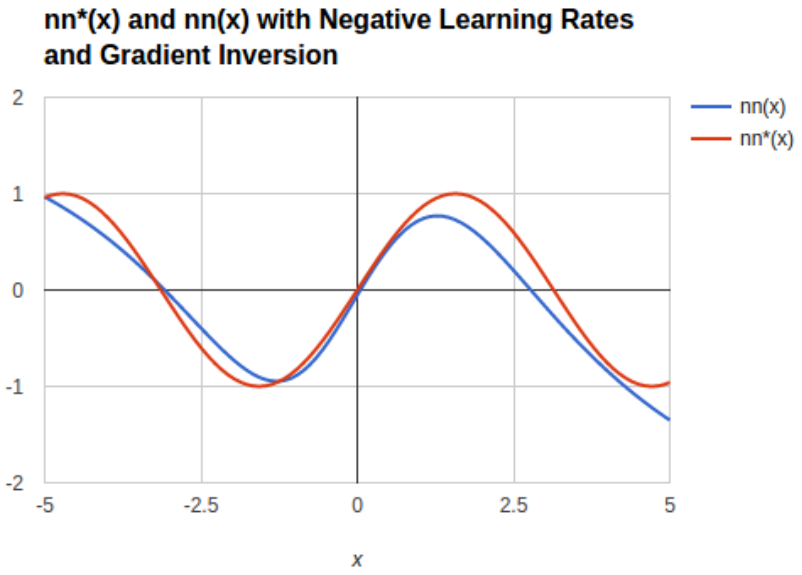

Figure 4: Output with per example learning rates $r_i = -\frac{dist^*(x_i,z_i) - \frac{\sum_j z_j}{n}}{max_i(|dist^*(x_i,z_i) - \frac{\sum_j z_j}{n}|)}$. Incoming gradient is inverted on negative examples.

It looks like we have solved our problems with this method. We can compare, in figure 5 the solution found by our network trained using learning rates to a network trained directly from examples drawn from the target function. The traditionally trained comparison network has exactly the same architecture as our network trained using the learning rate as a training channel. It was presented with the same example inputs, in the same number of training iterations. The solution found is virtually identical to the solution found with traditional training.

4 What We Are Really Doing

Now let us take a step back and see what we are really doing. We are training a function approximator with random inputs and outputs. We only give the function approximator a scalar signal that corresponds with how "good" a given example is. We are doing reinforcement learning.

Reinforcement learning typically takes on two forms, learning with an environment model or learning with policy evaluation.

If we have a model of the environment, we can simulate what will happen when we take a particular action in a given state. Then we can explore the model and search for the action that increases the probability of getting rewards. For example, if the learner is playing checkers the learner can simulate thousands of possible games and choose the move that leads to the most winning games. This approach has recently been used to master the game of Go.[2]

Another approach finds the function $Q(s, a)$ where s is the current state and a is an action. The function $Q(s, a)$ gives the expected value of the state-action pair. The learner can then find the

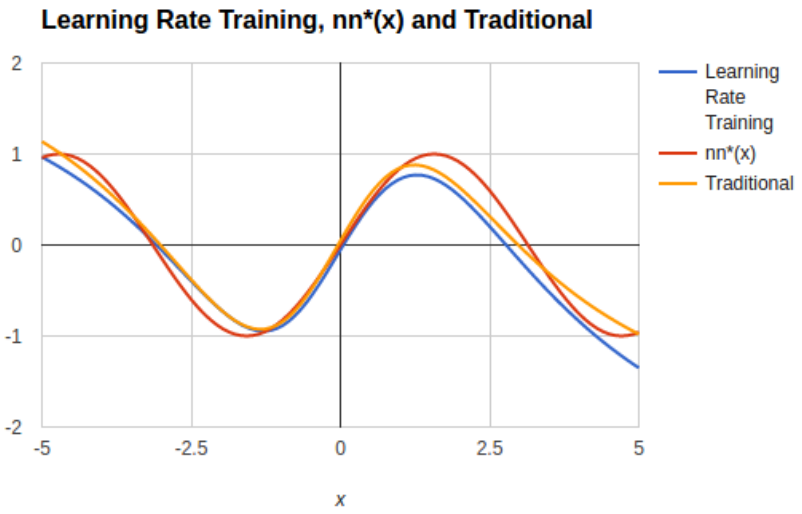

Figure 5: Comparison between network trained using learning rates and a network trained using correct examples.

action that maximizes Q. This is aptly called q-learning. Q-learning has been recently used to master several simple video games.[1]

Our method has several features that distinguish it from other reinforcement learning methods. First, there is no model of the environment. We do not need to simulate the outcome of actions. This method is totally "online". Secondly, we do not need to perform any policy evaluation. This is rather unusual.

Instead of learning the environment or the q-values, we directly learn a policy. Because **q**-learning learns **q**-values, and our methods learns a **p**olicy let us call it p-learning.

4.1 How to Use P-Learning as Reinforcement Learning

The reinforcement learning paradigm is that the agent receives the state of the environment s. It may then take action a, and receive reward r. The agent's goal at time t_0 is to maximize the total discounted future rewards $\sum_t \gamma^{t-t_0} r_t$ with some discount factor $\gamma \in [0, 1]$.

Before we have learned a good policy, we must explore the environment. We can do this by applying random moves to the environment.

We should do this until we have a good set of experiences. Experiences are action-state-reward triplets (a, s, r). For each triplet, we must pre-back-propagate the discounted rewards. That is, propagate backwards the discounted reward from each experience that receives a reward.

When we receive an experience at time t_0 where $r_{t_0} > 0$ we replace each other experience's reward as $r_t \leftarrow r_t + \gamma^{t_0-t} r_{t_0}$. We can stop the reward propagation when we know that the previous states did not affect the current state being updated. This might happen if we know that a state was at the start of a game in a string of games. We should not propagate rewards across games.

After reward propagation, construct a set of training examples where each example e_t corresponds to the experience at time t. Each example looks like

$$e_t = (s_t, a_t, \frac{r_t - \frac{\sum_j r_j}{n}}{max_i(|r - \frac{\sum_j r_j}{n}|)}).$$

We can then train a function approximator using gradient descent using p-learning on our set of training examples. The actions that received or lead to high rewards are trained with high learning rates. For actions that received the lowest rewards, the approximator is trained to avoid using negative learning rates. Actions that lead to average rewards do not get trained; they have learning rates close to zero.

Please note that actions that received average rewards have a very low learning rate. These can be safely dropped from the training set. When the environment leads to many rewards of close to average value, this can speed up training significantly. For example, if the agent plays many games that end in a draw while winning and losing an equal number of games, we can simple ignore all the draw games. We can do this by removing all experiences from the training dataset where the reward for that experience is within some small value of the mean of all the rewards from all experiences.

We have tested p-learners in simple "mouse and cliff" games, where the agent must move on a checkers board from a start tile to a goal tile while avoiding hazard tiles. P-learners learn comparable policies, in the same number of games, as q-learners where the Q function is approximated by a neural network with the same architecture as the p-learner's policy net.

5 Conclusion

In this paper we have shown how to use negative examples to train function approximates in regression tasks. We have also shown how to use negative learning rates to effect this training. We also have introduced a method of reinforcement learning, p-learning, that directly learns a good policy without learning either an environment model or doing any policy evaluation.

Acknowledgments

Thank you for reading :)

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., . . . Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.

[2] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Driessche, G. V., . . . Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), 484-489.